An Implementation of Vidhi in IF—Prolog

by

SUNIL SOMAN NAIR

DEPARTMENT OF MECHANICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

APRIL, 1995

An Implementation of Vidhi in IF–Prolog

A thesis submitted
in partial fulfilment of the requirements
for the degree of

Master of Technology

by

Sunil Soman Nair

to the

Department of Mechanical Engineering
Indian Institute of Technology, Kanpur
April 1995.

Certificate

This is to certify that the present work on **"An Implementation of Vidhi in IF–Prolog"**, by Sunil Soman Nair, has been carried out under our supervision and has not been submitted elsewhere for the award of a degree.

Rajeev Sangal

Dr. Rajeev Sangal,
Professor,
Department of Computer
Science & Engineering,
Indian Institute of Technology,
Kanpur.

Keshav Kant
24.4.95

Dr. Keshav Kant,
Associate Professor,
Mechanical Engineering
Department,
Indian Institute of Technology,
Kanpur.

ABSTRACT

An expert system shell is a system that has provision for the efficient storage and retrieval of assertions in the database, an appropriate user interface, and an ability to pose questions to the user. *Vidhi* is an expert system shell which was first implemented in LISP. A later implementation was available in IF/Prolog. Prolog's built-in inference mechanism simplified the implementation of the shell.

Subsequent to this, several modifications have been made to this IF/Prolog shell. These mainly consist of a more efficient and rigorous implementation of functional dependencies. The result of this is that *Vidhi* in IF/Prolog has become more robust. The expert system that has been implemented is for the thermal design of wet cooling towers. Both natural and mechanical draft wet cooling towers of the counterflow and crossflow type can be designed using this system. A user interface has been implemented in IF/Prolog for the cooling tower system. The user interface is necessary so that the user can interact properly with the system and the design parameters can be extracted and printed out for the user's convenience.

Vidhi in IF/Prolog was available for use on the HP system. Further attempts to port the shell required modifications to it. The popularly available version of Prolog on PCs is Turbo Prolog. *Vidhi* in IF/Prolog thus had to made compatible with Turbo Prolog. Since many of the standard predicates of Prolog are not available in Turbo Prolog, this required making use of the Prolog Inference Engine which is available with the Turbo Prolog compiler (version 2.0).

Although the shell itself could be ported without difficulty, the same could not be done for the database required to implement the cooling tower system. This was largely due to insufficient conventional memory being available on a PC. It was however possible to run the expert system to provide for partial designs.

Acknowledgements

I would like to express my gratitude to Dr. Keshav Kant for his guidance during the course of this work. My thanks are also due to Dr. Rajeev Sangal who helped me to understand **Vidhi** and who was kind enough to give me some of his time in spite of having a very busy schedule.

I would also like to thank Gokul Rajaram and S.S. Sastry for the assistance they provided. S.S. Sastry was always willing to promptly answer my many queries.

Sunil S. Nair

Contents

Chapter 1

Introduction

1.1 The Expert System

An expert system consists of an inference engine, a knowledge base, a knowledge base editor and an explanation module. An expert system shell is a package that consists of all the above components except the knowledge base. An expert system besides having the ability to solve problems must possess the following characteristics:

1. It should engage in a dialogue with the user to acquire the relevant details of the problem.

2. It should be able to explain its problem solving process.

3. It should be easily modifiable to take care of new developments.

4. It should be able to deal with partial information. When all the required details to solve a problem are not available, it should still do the best that is possible. The solutions should degrade gracefully rather than suddenly and completely [8].

The power of an apparently intelligent program such as an expert system to perform its task well depends primarily on the quantity and quality of knowledge it has about that task. The knowledge of an expert system consists of *facts* and *heuristics*.

The facts constitute a body of information that is widely shared, publicly available, and generally agreed upon by experts in that particular field. The heuristics are mostly private and classified. They consist of little discussed rules of plausible reasoning that characterize expert level decision making in that field.

There is a basic need to keep the representation of the knowledge of a particular field separate from that of the logical mechanisms for processing and interpreting this knowledge. This ensures that the knowledge base can be modified at will, without the need of having to change the inference mechanism. Here we shall mostly be dealing with the logical mechanisms. The present *Vidhi* shell has been implemented in IF/Prolog. From now on we will use the term *Vidhi* to refer to the implementation of *Vidhi* in IF/Prolog, unless explicitly mentioned.

The present system is based on *backward chaining* where one begins with the goal and goes backward towards the facts. There is a mechanism in the shell which is used for eliciting information from the user. This is accomplished by introducing rules that cause questions to be issued. Such rules make use of a built-in predicate called 'ask-user'.

1.2 Programming in Prolog

As mentioned previously, the present version of *Vidhi* has been written in IF/Prolog. The standard features of Prolog are described in [3]. It supports all the predicates in [3] and a few others besides. IF/Prolog is ideally suited for the purposes of building a system as extensive as *Vidhi*. Building the database is relatively simple, and one can use all the features of Prolog except the 'cut'.

Programming in Prolog consists of

1. Declaring some *facts* about objects and their relationships.

2. Defining some *rules* about objects and their relationships.

3. Asking *questions* about objects and their relationships.

The Prolog system enables the computer to be used as a storehouse of facts and rules, and it provides ways to make inferences from one fact to another.

In Prolog, the names of all relatonships and objects must begin with a lower case letter. For example, *brother, quincey, jessica* . The relationship is written first, and the objects are written separated by commas, and the objects are enclosed by a pair of round brackets. For example, *brother(quincey,jessica).* The full stop "." must come at the end of a fact. In Prolog, rules are used when you want to say that a fact *depends* on a group of other facts. In English, we use the word "if" to express a rule. For example,

X is a sister of Y if :

X is female, and

X and Y have the same parents.

This can be written as the following Prolog rule.

```
sister_of (X,Y) :-
        female (X),
        parents (X,M,F),
        parents (Y,M,F).
```

Here X,Y, etc are *variables.* When Prolog uses a variable, the variable can be either instantiated or not instantiated. A variable is instantiated when there is an object that the variable stands for. Prolog can distinguish variables from names of particular objects because *any* name beginning with a capital letter is taken to be a variable.

When Prolog is asked a question containing a variable, Prolog searches through all its facts to find an object that the variable could stand for. Questions like these are what form *goals.* When we have more than one goal, we have the idea of the *conjunction* of goals. Conjunction is merely a way of saying *and.* To satisfy a conjunction of goals, Prolog attempts to satisfy the first goal. If the first goal is in the database, then Prolog will mark the place in the database,and attempt to satisfy the

second goal. If the second goal is satisfied, then Prolog marks *that goal's* place in the database, and we have found a solution that satisfies both goals.

It is important to remember that each goal keeps its own place-marker. If, however, the second goal is not satisfied, then Prolog will attempt to re-satisfy the previous goal (in this case the first goal). Remember that Prolog searches the database completely for each goal. If a fact in the database happens to match, satisfying the goal , then Prolog will mark the place in the database in case it has to re-satisfy the goal at a later time. But when a goal needs to be re-satisfied , Prolog will begin the search from the goal's own place-marker, rather than from the start of the database.

When handling a conjunction of goals, Prolog attempts to satisfy each goal in turn, working from left to right. If a goal becomes satisfied, variables previously uninstantiated might now become instantiated. If a variable becomes instantiated, all occurrences of the variable in the question become instantiated. Prolog then attempts to satisfy the goal's right-hand neighbour, starting from the top of the database. As each goal in turn becomes satisfied, it leaves behind a place marker in the database, in case the goal needs to be resatisfied at a later time. Any time a goal fails (a matching fact cannot be found), Prolog goes back and attempts to satisfy its left-hand neighbour, starting from its place marker. Furthermore, Prolog must "uninstantiate" any variables that became instantiated at this goal. The behaviour of Prolog, when it repeatedly attempts to satisfy and resatisfy goals in a conjunction, is called *backtracking.*

A special mechanism that can be used in Prolog programs is the "cut". The cut allows you to tell Prolog which previous choices it need not consider again when it backtracks through the chain of satisfied goals. There are two reasons why it may be important to do this:

1. The program will operate faster because it will not waste time attempting to satisfy goals that you can tell beforehand will not contribute to a solution.

2. The program occupies less of the computer's memory space because more economical use of memory can be made if backtracking points do not have to be

recorded for later examination.

In some cases, including a cut may mean the difference between a program that will run and a program that will not.

Syntactically, a use of cut in a rule looks just like the appearance of a goal which has the predicate "!" and no arguments. As a goal, this succeeds immediately and cannot be re-satisfied. It also has side-effects which alter the way backtracking works afterwards. The effect is to make inaccessible the place markers for certain goals so that they cannot be re-satisfied.

The formal definition of the effect of the cut symbol is as follows:

When a cut is encountered as a goal, the system thereupon becomes committed to all choices made since the parent goal was invoked. All other alternatives are discarded. Hence an attempt to re-satisfy any goal between the parent goal and the cut goal will fail.

From what has been said above , it should appear that *Vidhi* in Prolog is much more concise than the LISP version, and this is indeed so. Later on we will look at the details of the implementation.

1.3 The Cooling Tower System

The expert system that has been been implemented is for the thermal design of wet cooling towers as described in [2]. Both natural and mechanical draft wet cooling towers of the counterflow and crossflow type can be designed using this system. A cooling tower is a device which can be used to cool water by a simultaneous process of heat and mass transfer. The function of a wet cooling tower is to reduce the temperature of circulating water by bringing it into direct contact with an air stream.

The rules for the expert system are developed using a tree-like structure for the entire design process. The tree stucture is formed by the sub-goals which are generated in trying to satisfy the main goal. Such a tree formed during a computation will be dynamic.

1.3.1 The database

The database was constructed from experimental studies carried out to determine the various design parameters as well as the data available from the published literature on cooling towers. A comprehensive discussion is to be found in [6]

The design of a cooling tower first proceeds with the selection of the type of tower. This may either be a natural draft one or a mechanical draft one. Once this is done the type of flow – crossflow or counterflow – is decided upon. We also have to determine the water to air loading ratio, the tower floor area, and the type of packing – splash or cellular . For a mechanical draft tower, after deciding the type of draft, the overall height and the power required by the fan as well as its discharge rate are determined. For a natural draft tower, the tower base diameter and other dimensions of the tower including the total height of the tower and the total pressure drop across the tower are determined. The design also includes fixing the blowdown and the make-up water rate and selection of a water distribution system and the type of drift eliminator.

1.3.2 The user-interface

The user-interface is necessary so that the user can interact with the expert system shell and obtain a design of his choice with as little effort and repetition as possible. The user interface uses the partial results to determine the values of any variables that have been instantiated at any time during the inference process. These values are then printed out after either the success or the failure of the goal. A more elaborate description of the user-interface is to be found in a later chapter.

1.4 Porting Vidhi to a PC

Vidhi was initially written in IF/Prolog and was thus available for use on the HP system. Further attempts to port the shell required modifications to *Vidhi*. To allow the shell to be available on a PC required making it conform with Turbo-Prolog, which is the popularly available form of Prolog for PCs.

Turbo-Prolog, as opposed to the standard versions of Prolog which are described in [3], is a compiled language. Most versions of Prolog are interpreted languages. Turbo-Prolog also does not have many of the commonly used predicates such as *univ, functor,* == , etc. The approach which has therefore been followed is to use the Prolog Inference Engine (PIE) which has been provided with the Turbo-Prolog compiler, *version 2.0* . The PIE as it stood was not sufficient for the purposes of *Vidhi* and several extensions had to be made in order to implement *Vidhi.* The PIE is described in [10].

Although the shell itself could be ported without difficulty, the same could not be done for the database required to implement the cooling tower system. This was largely due to insufficient memory being available on a PC. However, it is possible to run the expert system to provide the design of some of the components of the cooling tower.

1.5 An Overview of the Literature

The principal reference for the shell *Vidhi* is [9]. Here the shell has been implemented in LISP. First the programs for pattern matching are developed. Then the resoloution principle of logic programming is explained. There are algorithms describing unification, generation of unique variables, and resolution. The main features of *Vidhi* are contained in the text. However, many of the finer details have not been worked out, but have been left as exercises. The complete version of *Vidhi* in FranzLisp had been implemented on the HCL system in the department of Computer Science and Engineering at I.I.T. Kanpur.

The Prolog version was the work of S.S.Sastry and Gokul Rajaram [7]. The knowledge base is described in [6]. So is the user-interface, although both have been written in Lisp. The methodology of the design of cooling towers is taken from [2], [1], and [4].

The general features of Prolog are to be found in [3]. The features of Turbo-Prolog are described in [10].

1.6 Organization of the Thesis

The introduction has provided a brief outline of the nature of the work done. Much of the work can only be clearly explained by actually describing the various algorithms that have been used to implement *Vidhi*. Subsequent chapters give a detailed idea of topics which have so far only been touched upon.

Chapter 2 deals with the first version of *Vidhi* in Prolog An outline is provided of the various component programs and the way the stack is manipulated. Functional-dependencies (FDs) are defined and the checks which are necessary to prevent their violation are explained. The program to ask the user questions and to record the user's responses also forms an essential part of the shell.

Chapter 3 deals with the modifications that have been made to *Vidhi*. These mainly involve the addition of certain functions to check the consistency of facts that have been inferred and more rigorous tests for FDs. With these modifications, *Vidhi* has become much more robust. The additions have made it a little slower, but the change has not been appreciable. The elaboration facility has also been implemented.

Chapter 4 deals with a description of the main features of the cooling tower system. Flow-charts have been provided which give an idea of the way the inference proceeds. The user-interface for the cooling tower has been described.

Chapter 5 deals with the porting of the shell to a PC. The Prolog Inference Engine has been described in brief and the main additions have been elaborated. Certain predicates have been changed and this has been explained. Finally the limitations in memory that have been a problem for the cooling tower system are analysed.

Chapter 2

Features of Vidhi

2.1 Introduction

An expert systems shell is a system that has provision for efficient storage and retrieval of assertions in the database, an appropriate user interface, and an ability to pose questions to the user when facts and rules fail. We match the goal with assertions in the database, which in general yields a new set of goals, and so on. This is called *backward chaining* , because one begins with the goal and goes 'backward' towards facts.

Vidhi is the expert systems shell described in [9]. The original version was implemented in LISP. LISP is a widely used artificial intelligence language whose name is derived from LISt Processor. The LISP version uses pattern matching and logic programming. Predicates are used to express the relationships between objects. An instance of this is found in the LISP expression (wbulb-temp bombay 27.8). Here wbulb-temp is the predicate and its arguments are bombay and 27.8 .

In LISP, formulas can take on truth values, and those which are known to be true are also known as assertions. An assertion with an antecedent is called a rule, while an assertion with an empty antecedent is called a fact. A formula whose truth value is to be determined is called a query or a goal formula. If the goal formula has a variable, then the binding of a variable for which the formula is true is the answer. If

the formula can't be proved true for any binding, then the answer is false. A suitable way is first found to represent the database of assertions. Each individual assertion is represented as a triple containing the consequent, antecedent and the name of the assertion. To speed up the access to the database, we store the assertions in the property list of the predicate in its consequent.

There are two rules of inference using which the truth value of a formula can be determined to give an initial set of assertions:

1. *Universal Instantiation:* Substituting a variable in a rule yields an assertion.

2. *Modus Ponens:* If the formulas in the antecedent of the assertion are true, then the consequent is also true. A formula is true if it is asserted to be true or if it can be inferred to be true from the rules of inference.

The inference algorithm of the shell takes any query and performs a series of operations. If the predicate currently involved is a computational function or predicate, then the arguments are passed on to the computational predicate or function after substituting the values for the arguments of the predicate. If the predicate is one of ask-user, then the actions that follow have been described in the next section. If the predicate is not one of the two former types, then the database is searched for the rules and facts stored for it. These are tried in the order in which they appear in the database.

The present version of *Vidhi* has been implemented in IF/Prolog, and many of the problems have been simplified as a result of Prolog's built-in inference mechanism.

2.2 Posing Questions – The Ask-user Predicate

In expert systems, not all the information is available in the database of assertions. Frequently the information has to be elicited from the user. Since not all possible questions need be asked of the user, these questions are asked depending on the goal and the user's former responses. Rules that cause questions to be asked make use of a built-in predicate called **askuser**.

askuser behaves differently from other predicates. The database is not searched to test the truth value of a formula involving **askuser**. Instead, a function with the same name is called that interacts with the user. If the user is unable to provide the answer, the formula is not true and it is recorded that the user does not know the answer. Otherwise the formula is satisfied and an appropriate fact is asserted.
The syntax of **askuser** is as follows:

askuser(<Fact>, <Question>, <Target Vars>, <Types for Target Vars>).

The 'Fact' indicates as to what will be known if the question is successfully answered. If this is 'none', then nothing is asserted if the question succeeds or fails. This means that this question will be repeated each time 'askuser' is invoked. If this 'Fact' is not 'none', then the question is posed only once, which is when 'askuser' is first invoked. The question is a list of strings and variables interspersed with each other. However, these variables should be instantiated to some value for the question to be asked. The target variables must also not be instantiated. The types supported for target variables in the IF/Prolog version are:

1. universal: no restriction on type.
2. int: integer.
3. float: real number.
4. string: a string.
5. boolean: yes or no and true or false.
6. range (<lower>, <upper>) : specifies the range of the number.
7. tolerance (<mean>, <variation>) : here the range is (<mean> – <variation>) and (<mean> + <variation>).
8. enumerate ([<v1>, <v2> ....]) : a list of values that the answer might take.
9. pred (<type-pred>) : this specifies the name of the user-defined predicate to be applied to the value to test its type.

Instances of the use of askuser are:

1. askuser (temp (X,Y),
   ['What is the temperature of', X, 'in degrees Celsius'],
   [Y],
   [range (0,100)]).

2. askuser (none,
   ['Input a number'],
   [N],
   [integer]).

3. askuser (none,
   ['Input an even number'],
   [N],
   [pred (checkEven)]).

A number of responses are possible when the question is put to the user. The type of responses possible are displayed if the user types ‘help’. These are enumerated below:

1. dontknow / dunno: If the answer is not known.
2. repeat: To see the question again.
3. what: To see the elaboration provided for the question.
4. type: The type of valid responses for the question.
5. system (<cmd>): To execute any top level command <cmd>.
6. help: To see this message.
7. yes,no,true *or* false: When the question expects a confirmation or otherwise.
8. <values>: As many as needed by the question, separated by spaces.

The elaboration facility was not provided initially. The predicates required for interaction with the user are included in the file ‘ask.pro’.

2.3 Storing Intermediate Results

If all inferences made by the system are stored, they will occupy space and will also increase the search time in the database. On the other hand, an inferred formula if available directly will save the time needed to infer it. The decision to store inferences must thus be made judiciously. The shell depends on the user's advice about what inferred formulas to store. Such predicates are declared 'intermediate' predicates. Inferences involving intermediate predicates will be stored in the database and all future queries on this predicate will return values from these stored inferences. This may become a problem because attempts to resatisfy the query will only occur if there are more 'intermediate' results stored about this predicate. In other words, the remaining rules are not looked at for further inference.

A predicate <pred> is intermediate if it occurs in the database as

intmd (<pred>, <arity>).

There are certain built-in predicates in *Vidhi* for dynamically declaring and 'un-declaring' predicates to be intermediate. The syntax for declaring a predicate as intermediate is

definter ([intmd (<pred1>, <arity1>), ...]).

The syntax for undeclaring a predicate as intermediate is

undefinter ([intmd (<pred1>, <arity1>), ...]).

After a predicate is declared to be intermediate, any inference on this predicate is asserted in the database as $$$interm (Inference). We have to prevent these intermediate inferences from being mixed with normal facts. If such mixing takes place, then not only are duplicate answers produced, but every duplicate answer is produced only after going through the entire inference process. This is of course absurd. As mentioned before, once a goal matches with any intermediate result, then further inferences on the basis of rules are precluded.

In the case of question rules, storing inferences has an effect that goes beyond efficiency. Storing them ensures that if a question has been answered affirmatively,

it is not asked again. This avoids repeated questioning of the user for the same information. The only case when this is not done is when 'Fact' is none. This has been explained before. For a successful answer, $$$ans (Fact) is stored in the database. If the user answers 'dontknow' or in the negative, then $$$false (Fact) is asserted (and askuser fails).

2.4 Functional Dependencies

It often happens that the value of one argument of a predicate can be functionally determined by the value of the other arguments. This notion can be captured by the use of *functional dependencies.* Functional dependencies are specified in *Vidhi* as follows:

fd (<pred>, <arity>, <Independent ArgList>, <Dependent ArgList>).

For instance, fd (student, 2, [1], [2]) is taken to mean that the first argument of student (arity 2) functionally determines the second argument. The first argument could be the roll number and the second argument could be the name. Any subset of the arguments of a predicate can be declared to be independent and any other subset can be declared dependent. Of course, these subsets must be mutually exclusive.

There are certain built-in predicates provided for dynamically declaring and removing functional dependencies during a session. These are:

deffds (<list of fds>).

undeffds (<list of fds>).

Examples of the use of these are

a) deffds ([fd(student, 2, [1], [2]), fd(course, 2, [1], [2])]) and

b) undeffds ([fd(student, 2, [1], [2]), fd(course, 2, [1], [2])])

There are two advantages in declaring FDs on predicates in a database.

1. It allows inconsistencies to be detected.

2. It can speed up inference. When we try to infer a goal and find an assertion that matches on the arguments that functionally determine some of the other arguments, and there is a disagreement on the dependent arguments, we say that a counter-example has been found. The goal fails and no further search is made in trying to infer it. FDs can also be used to set up intelligent backtracking.

2.5 Miscellaneous Predicates

This section provides a brief description of some predicates used in *Vidhi*. These have been implemented in order to provide a simpler and more convenient syntax.

1. atoi(<string>, <integer>): This predicate converts a string to an integer. We assume that the input string represents an integer.

2. atof(<string>, <float>): This predicate converts a string to a real number. The string should represent a real number in decimal point notation. The scientific notation is not supported by this predicate.

3. asserta(<pred>), assertz(<pred>): These assert <pred> at the beginning and at the end of the database respectively. Before asserting, they check whether this predicate is consistent or not with the existing facts of the database.

4. quit: This ends the session with *Vidhi*.

5. restart_session: This clears the database of all the intermediate results and answers provided by the user. However, all the files loaded since the beginning of the session will still be present in the database.

6. load (<file-name>): This loads <file-name> into the database. While loading it checks for consistency of the facts and informs the user of any inconsistency.
   Other mechanisms for loading files are :
   consult (<file-name>),
   [<file-name1>, ..., <file-nameN>]
   reconsult (<file-name>),

[-<file-name1>, ..., -<file-nameN>]

These predicates differ from ‘load’ in that they do not carry out any checks for the consistency of the database.

2.6 The dfs predicate

In this section we will look at some of the implementation details of *Vidhi*. The inference mechanism, as has already been stated, is that of ‘backward-chaining’. It is based on Prolog’s inference mechanism. The code for carrying out the inference and checking consistency is contained in the file ‘dfs.pro’. The principal predicate here is called ‘dfs’ and its syntax is as follows:

dfs (<List of Goals>, <List of Inferences Made>).

The first argument is the list of goals to be satisfied, the second is the list of all the inferences made till the present time, i.e. those inferences made along the path from the root goal to the present goal. This list is necessary to implement the functional dependencies.

If any goal has an FD associated with it, unification is attempted with earlier inferences on the basis of the independent arguments of the FD. If this succeeds, then the children goals for this predicate are discarded. This is because for this predicate (having these independent arguments), there cannot be any other inference. If no unification results, then the children goals are retained. If unification fails, then the inference fails because this means that this inference violates some FD.

If the goal is ‘askuser’, then before posing the question checks are made to ensure that the question has no uninstantiated variables and that the target variables are all instantiated. All the computational predicates of Prolog are supported. These are ‘+’, ‘−’, ‘*’, ‘/’, ‘mod’, ‘//’(integer division), ‘trunc’, ‘exp’, ‘^’, ‘ln’, and ‘abs’. Although using Prolog’s inference mechanism makes the task of satisfying goals easy, one cannot have explicit control over the stack. This means that the parent goal (or even any ancestor goal) is not accessible. This is the reason why ‘Fact’ has to be one of the arguments of ‘askuser’.

The forming of children goals and their subsequent resolution is performed by the following 'dfs' predicate:

```
dfs([Head | OtherGoals], InferredL) :-
        not(clause($$$interm(Head), true)),
        clause(Head, Body),
        set_global(done, false),
        checkInferConsistency(Head, InferredL),
        (
                get_global(done, true),
                !,
                dfs(OtherGoals, InferredL)
        ;
                get_global(done, false),
                formNewGoals(Body, OtherGoals, NewGoals),
                storeIntermediate(Head),
                dfs(NewGoals, [Head | InferredL])
        ).
```

If a predicate has been declared as 'intermediate', then any inference for this predicate would have been stored in the database as $$$interm (inference). If a fact of this form exists, then the inference need not be carried out. *checkInferConsistency* checks if 'Head' is consistent with the inferences made until now and which are stored in 'InferredL'. *set_global* and *get_global* are predicates for setting up and obtaining global variables in IF/Prolog and are defined in [11]. *formNewGoals* forms a list of new goals by appending the children goals to the head of the old goal list. *storeIntermediate* stores all inferences for those predicates which have been declared as 'intermediate'. Intermediate results are naturally stored only if the inference is not already a fact in the database. Several improvements have been made on this 'dfs' predicate and these have been described in detail in the next chapter.

Chapter 3

Modifications to the Shell

The previous chapter contained a discussion of the way goals are resolved by the *Vidhi* shell. The boundary condition for the 'dfs' predicate is **dfs**([ ], _). This is the case when all goals have been successfully resolved. The negation of a goal i.e. not(Goal) is decided by checking if Goal can be satisfied. If it is successfully satisfied, then not(Goal) fails. There is no need to check for violation of FDs in case not(Goal) is true, because FDs may only be violated by positive goals. The 'dfs' predicate however requires modification when we have to satisfy positive goals and the children goals which are spawned by parent goals. These modifications are mainly in the nature of additions. They form the subject matter of the folllowing sections.

3.1 The modified dfs predicate

The 'dfs' predicate which forms children goals and tries to resolve them was described in the last chapter. The modifications mainly concern the way the goals travel through this predicate. So we shall concentrate on this particular 'dfs' predicate. This predicate is now:

```
dfs([Head | OtherGoals], InferredL) :-
        not(clause($$$interm(Head), true)),
        shouldIresatisfy(Head),
        !,
        clause(Head, Body),
        set_global(matchedAllfds, false),
        checkInferConsistency(Head, InferredL),
        get_global(matchedAllfds, Done),
        (
            Done = true,
            get_global(resatisfy, Resat),
            (
               Resat = false,
               !
            ;
               Resat = true
            ),
            dfs(OtherGoals, InferredL)
        ;
            Done = false,
            formNewGoals(Body,OtherGoals, NewGoals),
            dfs(NewGoals, [Head | InferredL]),
            checkInferConsistency(Head, InferredL),
            storeIntermediate(Head)
        ).
```

shouldIresatisfy(Head) is used to check if all the 'key' arguments are bound. This is used to set the value of *Resat*. *checkInferConsistency* is used to check if the *Head* is consistent with the earlier inferences and facts. This check is done only if all the independent arguments of all the FDs are bound. Else such a check is obviously not necessary. *checkInferConsistency* has also undergone changes and is dealt with later.

The counter *Done* is set to *true* if a match is found on all the independent arguments of all the FDs. Such a match is done using the earlier inferences and facts present in the database. If a match is found, then there is no need to satisfy the children of this goal. The 'cut' is present after *Resat = false*, because if all the key arguments are bound (in which case *Resat* is set to *false* in *shouldIresatisfy*), then there is no need to resatisfy the goal because each such 'resatisfying' will yield the same result. *checkInferConsistency* appears twice in the above predicate. This is needed because it may happen that the first check is not carried out. In this case, the second check is required so that any inconsistency which escaped the earlier check may be revealed.

3.2 Rigorous tests for fds

The purpose of the predicate *shouldIresatisfy* is to ensure that a goal is resatisfied only for the following two cases:

1. All the 'key' arguments of the goal are not bound.

2. All the arguments which are not part of any FD are not bound. We shall call such arguments 'independent arguments'.

For any predicate we will store the key arguments as $$$key(...) and the independent arguments as $$$indep(...). Of course, such considerations apply only when there are FDs associated with a predicate. Otherwise we shall always try to resatisfy our goal. A complete description of the *shouldIresatisfy* predicate is given on the following page :

```
shouldIresatisfy(Head) :-
        functor(Head, Pred, Arity),
        functor(X, Pred, Arity),
        not(clause(X, Body)),
        !,
        fail.

shouldIresatisfy(Head) :-
        getFDlist(Head, FDlist),
        FDlist = [ ],
        !,
        set_global(resatisfy, true).

shouldIresatisfy(Head) :-
        getKeyIndep(Head, Key, Indep),
        bound(Key),
        bound(Indep),
        !,
        set_global(resatisfy, false).

shouldIresatisfy(_) :-
        set_global(resatisfy, true).
```

The first rule is used to check if *Head* corresponds to any rule in the database. *getKeyIndep* computes the 'key' arguments for a predicate. This is done taking into account only those arguments which are part of some FD. *getKeyIndep* is described on the next page:

```
getKeyIndep(Head, Key, Indep) :-
        Head =.. [Pred | Args],
        getKeyArgs(Head, KeyArgs),
        getargs(Args, KeyArgs, Key),
        getIndepArgs(Head, IndepArgs),
        getargs(Args, IndepArgs, Indep).

getKeyArgs(Head, KeyArgs) :-
        functor(Head, Pred, Arity),
        clause($$$key(Pred, Arity, KeyArgs), true),
        !.

getKeyArgs(Head, KeyArgs) :-
        functor(Head, Pred, Arity),
        formset(1, Arity, Set),
        getIndepArgs(Head, IndepArgs),
        filter(Set, IndepArgs, Nset),
        getFDlist(Head, FDlist),
        computeKey(FDlist, Nset, Nset, Nset, KeyArgs),
        assertz($$$key(Pred, Arity, KeyArgs)).
```

Nset is the list of all the arguments which participate in some FD or the other. The 'key' is computed by eliminating each argument one by one and checking if the 'closure' of the remaining arguments covers all the arguments of the predicate (which has FDs associated with it). *computeKey* is defined as follows:

```
computeKey(_ , _ , [ ], Key, Key).

computeKey(FDlist, AllArgs, [H | T], TempKey, Key) :-
        filter(TempKey, [H], Nkey),
        closure(FDlist, Nkey, Cl),
        (
          equalSets(Cl, AllArgs),
          !,
          computeKey(FDlist, AllArgs, T, Nkey, Key)
        ;
          !,
          computeKey(FDlist, AllArgs, T, TempKey, Key)
        ).
```

The 'closure' of a set X is computed as follows. For each FD: $A \Longrightarrow B$ we first check if A is a subset of X. If so, then $X = X \cup B$. If there is any change in X after we go through all the FDs, we repeat the above procedure. Otherwise X is the closure. The functions that perform these operations are:

```
closure(FDlist, X, Cl) :-
        aux_closure(FDlist, X, Y),
        (
          equalSets(X, Y),
          !,
          Cl = X
        ;
          closure(FDlist, Y, Cl)
        ).
```

```
aux_closure([ ], X, X).

aux_closure([fd(_ , _ , I, D) | Tail], X, Z) :-
        subset(I, X),
        !,
        union(X, D, Y),
        aux_closure(Tail, Y, Z).

aux_closure([_ | Tail], X, Z) :-

        aux_closure(Tail, X, Z).


subset([ ], X).

subset([H | T], A) :-
        member(H, A),
        subset(T, A).


union(X, [ ], X).

union(X, [H | T], [H | Y]) :-
        not(member(H, X)),
        !,
        union(X, T, Y).

union(X, [_ | T], Y) :-

        union(X, T, Y).


equalSets(X, Y) :-
        subset(X, Y),
        subset(Y, X).
```

The predicate *getIndepArgs* in *getKeyArgs* computes the set of what we have chosen to call 'independent arguments'. Remember that these are arguments that are not part of any FD for the predicate in question.

```
getIndepArgs(Head, IndepArgs) :-
        functor(Head, Pred, Arity),
        clause($$$indep(Pred, Arity, IndepArgs), true),
        !.

getIndepArgs(Head, IndepArgs) :-
        getFDlist(Head, FDlist),
        functor(Head, Pred, Arity),
        formset(1, Arity, Set),
        filterSet(Set, FDlist, IndepArgs),
        assertz($$$indep(Pred, Arity, IndepArgs)).
```

The function *formset* forms the set of integers from *1* to *Arity*. The function *filterSet* is defined below.

```
filterSet(Set, [ ], Set).

filterSet(Set, [fd(_ ,_ , A, B) | T], Nset) :-
        filter(Set, A, Set1),
        filter(Set1, B, Set2),
        filterSet(Set2, T, Nset).

filter([ ], _ , [ ]).
```

```
filter([H | T], A, Out) :-
        member(H, A),
        !,
        filter(T, A, Out).

filter([H | T], A, [H | Out]) :-

        filter(T, A, Out).
```

From the set *Set* we remove all the elements which are present in the sets *A* and *B*. This uses the predicate *filter* which is also defined above. *filter* does the actual filtering of one set on the basis of another set.

The new definition of *checkInferConsistency* is:

```
checkInferConsistency(Inference, InferList) :-
        getFDlist(Inference, FDlist),
        FDlist \= [ ],
        !,
        functor(Inference, Pred, Arity),
        make_fact_list(Pred, Arity, FactL),
        set_global(matchedAllfds, true),
        check_inference(Inference, FactL, FDlist),
        get_global(matchedAllfds, X1),
        set_global(matchedAllfds, true),
        check_inference(Inference, InferList, FDlist),
        get_global(matchedAllfds, X2),
        or(X1, X2).

checkInferConsistency(Inference, _).

or(true, _) :- set_global(matchedAllfds, true), !.
or(_ , true) :- set_global(matchedAllfds, true), !.
or(_ , _) :- set_global(matchedAllfds, false).
```

getFDlist gathers together all the FDs relating to the functor of *Inference*. The argument *FDlist* below should not be bound.

```
getFDlist(Inference, FDlist) :-
        functor(Inference, Pred, Arity),
        clause($$$fdlist(Pred, Arity, FDlist), true),
        !.

getFDlist(Inference, FDlist) :-
        functor(Inference, Pred, Arity),
        getall(fd(Pred, Arity, _, _), [ ], FDlist),
        FDlist \= [ ],
        !,
        asserta($$$fdlist(Pred, Arity, FDlist)).

getFDlist(_, [ ]).
```

check_inference checks that *Inference* is consistent with the given list of facts and inferences. It also checks if this inference is a member of the list. If it is so, then it is not added to the new inference list.

```
check_inference(_, _, [ ]).

check_inference(Inference, List, [fd(Pred, Arity, X, Y) | FDtail]) :-
        set_global(done, false),
        inference_agrees(Inference, List, X, Y),
        checkIfMatchedAll,
        check_inference(Inference, List, FDtail).
```

```
checkIfMatchedAll :-
        get_global(matchedAllfds, And),
        get_global(done, Done),
        and(And, Done).

and(true, true) :- !.
and(_, _) :- set_global(matchedAllfds, false).
```

matchedAllfds indicates whether *Inference* has matched on the independent arguments of all the FDs. This must be checked after *inference_agrees* is called for each FD. *checkIfMatchedALL* is present for this purpose. It computes the logical *and* of the *done* values for all the FDs. *inference_agrees* has not undergone any change.

3.3 The elaboration facility

For providing elaborations, an extra argument has been added to askuser. This has been called *Mark*. When $Mark = 0$, this means that no elaborations are provided, and the user is notified about this. Usually *Mark* has the same name as that of the predicate about which we provide the elaboration. The *elaborate* predicate in the database has two arguments. The first is the name of the predicate about which the elaborations are provided. *Mark* must be instantiated to this value. The second argument is the elaboration text.

The new syntax of **askuser** is :

askuser(<Fact>, <Mark>, <Question>, <Target>, <Types>).

3.4 Getting partial results

While the inference is going on, the sibling goals and the children goals are not differentiated and are kept in a single goal list. One needs to store a parent goal as soon its children goals are satisfied. If one separates sibling goals and children goals, it will mean storing much more information, and the inference mechanism will

get considerably slowed down. The procedure that has been followed here is to put *asserta($$$partial(Head))* at the end of the children goals. As soon as the children goals are satisfied, this goal will be asserted. With this modification, 'dfs' is now:

```
dfs([Head | OtherGoals], InferredL) :-
      not(clause($$$interm(Head), true)),
      shouldIresatisfy(Head),
      !,
      clause(Head, Body),
      set_global(matchedAllfds, false),
      checkInferConsistency(Head, InferredL),
      get_global(matchedAllfds, Done),
      (
          Done = true,
          get_global(resatisfy, Resat),
          (
             Resat = false,
             !
          ;
             Resat = true
          ),
          dfs(OtherGoals, InferredL)
      ;
          Done = false,
          formNewGoals(Body,[asserta($$$partial(Head) |
                         OtherGoals],NewGoals),
          dfs(NewGoals, [Head | InferredL]),
          checkInferConsistency(Head, InferredL),
          storeIntermediate(Head)
      ).
```

Chapter 4

The Cooling Tower System and the User Interface

4.1 Introduction

We are concerned here only with the wet counterflow and crossflow type of mechanical and natural draft cooling towers. Many factors influence the selection of the type of tower. Many of them are very general considerations. In the present expert system, a simple but effective selection process has been incorporated which is based upon some of the major differences existing between the mechanical draft cooling tower (MDCT) and the natural draft cooling tower (NDCT).

For very large installations, NDCT get the preference. NDCT are selected for power plants having a capacity of 500 MWe and above. The initial investment on a MDCT is much lower than that on a NDCT. However the total operation and maintenance cost favours a NDCT. Design accomodation, restrictions on tower dimensions, orientation with the direction of the prevailing wind and added capacity because of recirculation can boost tower cost in the case of a MDCT.

Choice of flow type in a cooling tower depends on several factors. Recirculation of outlet air, maintenance cost and cooling per unit volume of the tower are considered in order to select the proper type of flow. A crossflow tower is thermally less efficient

due to recirculation of outlet air. In a crossflow tower, an insufficient pressure head on the distribution pans also clogs the orifices because of algae and other debris that often collect in the system [5]. Crossflow towers therefore require higher maintenance cost than the counterflow towers. A counterflow tower produces more cooling per unit volume for less cost than a crossflow tower under the same conditions.

Such are the general considerations that help us to choose the type of cooling tower and the type of flow. After this is done, we have to evaluate several of the performance characteristics of the tower and the tower dimensions. The next section deals with the detailed steps necessary for the design.

4.2 The Design Steps

In the final design of the cooling tower, the following steps are to be followed [6]:

1. Decide upon the type of cooling tower. As mentioned above, this involves a choice between mechananical draft and natural draft towers. Subsequently, the type of flow — counterflow or crossflow — is also decided.

2. Determine water to air loading and the tower characteristics after knowing the inlet and outlet properties of air and water.

3. Determine water loading and air loading and estimate the tower floor area.

4. Decide upon the type of packing to be used. This involves choosing between the splash and cellular type of packings. After this the number of decks of packing and the packed height is determined. So is the pressure drop across the packing.

5. For a mechanical draft tower we proceed according to the following steps:

   a) Determine the overall height of the tower.

   b) Decide upon the type of mechanical draft. This involves choosing between forced and induced draft.

   c) Determine the power in HP required by the fan and also its discharge.

d) Determine the total static pressure of the fan.

6 For a natural draft tower we proceed as follows:

a) Choose the tower base diameter. It is often first found using an empirical relation.

b) Determine the other dimensions such as the total height of the tower.

c) Determine the total pressure drop across the tower.

d) If the user is satisfied with the dimensions obtained at this stage, then we go on to the next step. Else we go back to step **6(a)** to choose a new tower base diameter. The earlier data may be stored for comparison.

7. Determine the blowdown and the make-up water rate.

8. Select a water distribution system for the tower.

9. Select the type of drift eliminator.

The next page shows the flow-chart for the entire process.

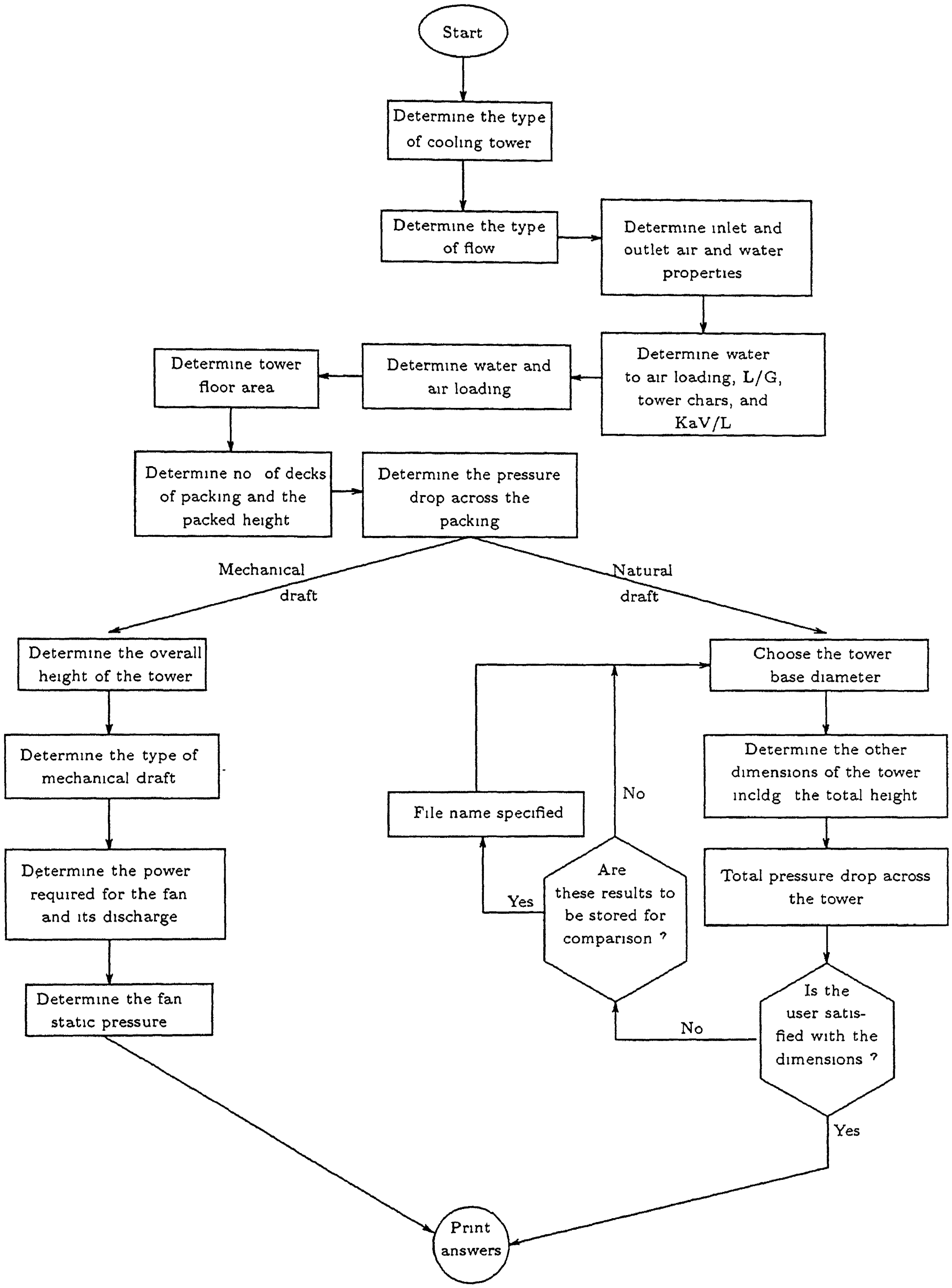

Fig 4.1 : Flow Chart for the Design of Cooling Towers

4.3 The user interface

The purpose of the user interface is to carry out the steps described in the last section in an efficient manner. Initially it proved impossible to obtain any partial results for reasons descibed in the previous chapter. With the assertion of $$$partial results, it is possible to retain the values of all the instantiated variables so that one may see just how far one has managed to proceed in the quest of getting a design.

The top level query in the cooling tower system is 'design(1)'. This has two rules. The selection of a rule depends upon the value of the predicate 'type-of-cooling-tower'. The user interface calls the 'dfs' predicate with the arguments design(1) and [ ] (the empty list). This list is built-up with the inferences that are made as the tree is traversed. The user interface takes over once the user types in 'hello' to the IF/Prolog prompt <?>.

The questions asked by the system are of two types:

- Mandatory questions
- Optional questions

Mandatory questions collect the essential information required to design the tower from the user. If a mandatory question is not answered by the user, the system cannot proceed further and the goal fails. Optional questions are not so critical. Even if the user responds with 'dontknow' to these questions, the design will still proceed In such cases a different path is tried by the system to resolve the query. Whether the overall design succeeds or not, the user interface will give us any partial answers that have been found.

The following are the mandatory question predicates:

1. installation
2. initial_investment
3. operation_and_maintenance_costs
4. recirculation_and_fogging

5. control_of_cold_water_temp

6. approach_to_wet_bulb

7. cooling_range

8. inlet_water_temp

9. town

10. packing_type

11. total_heat_load

12. dk_type (deck type)

13. pn_type (geometry of the packing)

There is a preliminary explanatory passage which is displayed if the user is new to the system. After this, if the user agrees to continue, then the design process begins. The user interface mainly consists of predicates to check whether the required design parameters have been instantiated or not. If they have been instantiated, then the values are displayed. If the user wants to save these values, then he can do so by specifying a file name. The user interface also makes sure the user does not exit the system until he wants to do so. The user interface predicates are contained in the file 'usrintfc.pro'.

4.4 Using the System

The *Vidhi* predicates are contained in the files 'dfs.pro' and 'ask.pro'. The database for the cooling tower is contained in the following three files:

1. ntow.pro

2. ftow.pro

3. tow.pro

The IF/Prolog interpreter can be invoked by simply giving the command 'vidhi'. The executable file 'vidhi' also sets the stack size and the trail size.

The database files can be simply loaded into the dynamic database using the command '[< file name >]'. If tests for FDs are to carried out, then one should invoke the 'dfs' predicate with the appropriate arguments. This has been described in an earlier chapter. The system may be started by typing in 'hello' to the IF/Prolog prompt. This must be done only after loading the file 'usrintfc.pro', which contains the user-interface.

Chapter 5

The PC Version

5.1 Introduction

In order to make *Vidhi* easily available, it was necessary to port the shell to a PC. The commonly available form of Prolog on a PC is Turbo Prolog, developed by Borland International, Inc. The features of Turbo Prolog are described in [10]. The built-in predicates of Turbo Prolog vary considerably from those of most Prolog derivatives. The 'core Prolog' described in [3] is the generally accepted standard. Turbo Prolog is a compiled language. This means that a Turbo Prolog program is written in a very different manner from say, an IF/Prolog program. Most derivatives of Prolog are interpreted languages. The Turbo Prolog program is divided into several 'sections'. Usually there is the *domains* section, the *predicates* section, the *goal* section, and the *clauses* section.

Turbo Prolog has six built-in domain types. These are characters, integers, real numbers, strings, symbols and files. The type of the domain must be indicated in the *domains* section of the Turbo Prolog program. As usual, predicates are used to represent data items as well as rules to manipulate the data. Predicates must be declared in the *predicates* section of the program. Not every Turbo Prolog program contains a goal. Some have external goals, which the user enters when the program starts. Turbo Prolog programs with external goals are interactive. The Prolog In-

ference Engine (PIE) described in the next section runs as an external goal. The purpose of using external goals is to allow the user to have free rein when using the data. When a program has no internal goal, the whole *goal* division, including the division header, is not included.

Turbo Prolog provides a *database* division for the declaration of dynamic database predicates. The database is called dynamic because variant clauses of the database predicate can be removed or new clauses can be added during the execution of the program. With a normal 'static' database in Turbo Prolog, database clauses are written into the program code. New clauses cannot thus be added during execution of the program. Another important feature of the dynamic database is that the database can be written to a disk file, or a disk database file can be read into memory.

Sometimes it is desirable to store part of the database in the program as a static database, and to assert the database information during the run of the program. The use of a memory resident database is usually appropriate if the database is of a limited size. The PIE runs like an interpreter and asserts all predicates in the dynamic database.

5.2 The Prolog Inference Engine (PIE)

The Turbo Prolog compiler does not allow the use of several of the built-in predicates of 'core' Prolog. These include many important predicates such as *functor, univ, arg,* == (for testing true equality), etc. In order to implement *Vidhi* on a PC, it is necessary to make these predicates available. The version 2.0 of Turbo Prolog has a Prolog Inference Engine available with it. The purpose of this PIE is to make available several of these missing predicates.

The PIE consists of the following five files:

1. **pie.pro** : This includes the user interface.

2. **pie.sca** : This is the scanner.

3. **pie.par** : This is the operator precedence parser.

4. **pie.inf** : This is the actual inference engine.

5. **pie.out** : This is the term writer.

To start the PIE, which acts much like an interpreter, one has to run the Turbo Prolog compiler, then load "pie.pro". The other files are automatically 'included'.

The module "pie.pro" consists of the declarations and the main program. It also includes the user interface. It defines both the parser and the scanner domains. The clause and operator database is included.

The module "pie.sca" is essentially a scanner which breaks up a string into a list of tokens of the following kinds: [ ], ( ), , variable strings (like List, Name, etc.), atoms (like hello, tom, etc.), integers, strings (like "Flaubert's parrot"), characters, comma, bar (|) and dot. Real numbers are not included, and therefore all real numbers which have to be read in must be converted using the Turbo Prolog type-conversion predicate *str_real(String,Real)*. Comments in the usual format (/* comment */) are accepted. Scanner errors may occur for integers larger than 32767 or less than -32768. This is dealt with in the next section. The symbols + , = , \\ , ? , − , : , ^ , @ , * , · , < , > , # , / , & , $ are reserved symbols and should not be used in predicate names.

The module "pie.par" implements an operator precedence parser. The priority and associativity of the operators are stored in the database predicate 'op'. In Prolog, operators can have three properties: a position, precedence class, and associativity. The position of an operator can be infix, postfix, or prefix. The precedence class is an integer whose range depends on each particular implementation of Prolog. The range used by most systems is from 1 to 1200. This precedence class is used to disambiguate expressions where the syntax of the terms is made explicit through the use of brackets. The associativity is to disambiguate expressions in which there are two operators in the expression that have the same precedence. We associate a special atom with an operator to specify its position and associativity. The possible specifiers for infix operators are xfx, xfy, yfx, yfy. Here we may think of f as representing an operator, and x and y as its arguments. The choice of x's and y's enables associativity

information to be conveyed. A y means that the argument can contain operators of the same or lower precedence class than this operator. On the other hand, a x means that any operators in the argument must have a strictly lower precedence class than this operator. The syntax of the predicate 'op' is **op(Prec, Spec, Name)**.

The module "pie.inf" implements the inference engine. The commonly used Prolog predicates have been implemented here. There are some differences however. Strings are not lists of integers. They form a separate type instead. Characters have their own type. Arithmetic operations are only implemented for integers, *get* and *get0* are not implemented, *read, readln*, and *readchar* are. *name* has not been implemented. Lists are not allowed as arguments to *functor* and *univ.* The '.' notation for lists is not accepted.

The module "pie.out" writes the output on the screen in a way that is easily understood. It displays terms, taking into due consideration the position of operators, the domain type and the insertion of brackets.

5.3 Additions to the PIE

In order to make the PIE useful for the purposes of an extensive shell like *Vidhi*, it was first necessary to implement arithmetic operations on real numbers. This required adding on several operator precedences in the database. Real number arithmetic has been implemented along with integer arithmetic. The predicate *is* automatically converts its arguments into real numbers. The final answer is also a real number This makes use of the type conversion predicate *int_real* which has been defined in terms of the Turbo Prolog predicates *str_real* and *str_int.* Care must however be taken to use this predicate only when one wants to convert an integer to a real number

The predicate *iis* is used only for integer arithmetic. Its most frequent use occurs when one wants to increment a counter. As mentioned before, one must be careful not to use integers below -32768 or greater than 32767. This will lead to a scanner error. In such cases, the integer should be converted from a string into a real number using the predicate *str_real.* The functions 'exp', 'ln', etc. were not implemented. Turbo

Prolog does not have the equivalent of the '^' function, but this could be defined in terms of 'exp' and 'ln'. The *name* predicate was rendered redundant because *readln* was available in the PIE. This made things easier where earlier it was necessary to manipulate lists of characters. The IF/Prolog built-in predicates for storing and retrieving global variables had to be replaced by functions for asserting these as facts in the dynamic database.

5.4 The Cooling Tower System on PC

Turbo Prolog makes use of several kinds of memory. The program code is stored in the 'code array'. The dynamic database is stored in the 'heap'. The three different run-time memory areas are the 'stack', the 'heap' and the 'trail'. The 'stack' is used for those functions which make use of tail recursion. *Vidhi* has several functions that make large use of tail recursion. Combined with a very large dynamic database such as that which is required for the cooling tower system, inadequacy of memory is a problem.

It is possible to first compile the PIE and create an executable file which when run will act as an interpreter. This saves the memory space occupied by the Turbo Prolog compiler. While compiling the PIE, one sets the stack size to as high a value as one feels is required. The trail size need not be very large. This leaves the remaining space free for the dynamic database. After loading 'dfs.pro' and 'ask.pro', it is possible to use the shell to obtain partial results only. However, by careful use of these partial results, such as the type of cooling tower required, the type of flow, etc., it is possible to obtain a good idea of the final design. Perhaps by using a newer version of Turbo Prolog, one could make use of the large extended memory available on all the newer PCs. This would enable the entire system to work.

5.5 Using the Shell

The PIE is activated by typing in the command 'pie' to the DOS prompt. The shell predicates are contained in the two files 'dfs.pie' and 'ask.pie'. These files are loaded into the database using the command 'consult('a: dfs.pie')' and 'consult('a: ask.pie')', assuming that these files are present in the drive 'a'. Now the inference mechanism is ready for use. The database files are now loaded, using the same command, and the 'dfs' is invoked with the appropriate goal list. This leads to questions being asked and the goals being resolved.

5.6 Epilogue

The system is quite interactive and user friendly. The IF/Prolog version can be used to effectively design cooling towers of practical use. The PC version, not being complete, can be used for the purposes of instruction and demonstration. Before using the system, one should collect all the information required to answer the mandatory questions. As much of the optional information available should also be obtained. This may often be present in the form of limitations on the design.

5.6.1 Limitations of the Present Work

The limitation of the cooling tower system is mainly due to an insufficient database which requires additional data for further improvement. The shell *Vidhi* is also open to further development. The main problems are:

1. Experimental data on different types of packings and drift eliminators is not available. Such data is usually classified by the industry.

2. The present system considers the case of wet natural and mechanical draft towers only. Dry towers, atmospheric towers and spray-filled towers are not considered.

3. The present system lacks the provision for changing an already assigned value of a particular design parameter. This is because *Vidhi* does not allow us to declare data dependencies.

4. The PC version, as has already been said, cannot design the tower completely because of memory limitations. A way should be found to get around this problem.

5.6.2 Conclusion

By implementing *Vidhi* in IF/Prolog, it has become possible to make the shell available to a large number of users. The earlier version's usefulness was severely limited as a result of the limitations of the system on which the shell was implemented. The IF/Prolog version carries out more stringent tests for FDs and has therefore made *Vidhi* more widely applicable. With the availability of newer versions of the Turbo Prolog compiler which can avail themselves of extended memory, it will be possible to have the expert system available over a wide range of PCs. At present the shell itself is available on PC, although there are problems in manipulating large databases. Having the system available on PC will enable it to be widely used both in the industry as well as for educational purposes. The shell may be used in various other areas where there is a need to design complex systems. Some areas where such a shell may be used have been described in [9].

The database has been created mostly from experimental and theoretical data. As mentioned in [6], the cooling tower system should first be tested in the field before being put to commercial use. It can be used for teaching purposes. Further work should concentrate on making the entire system available on PC.

Bibliography

[1] Berman, L.D., (1961), *Evaporative Cooling of Circulating Water,* Pergamon Press, New York.

[2] Cheremisinoff, N.P., and Cheremisinoff, P.N., (1981), *Coolıng Towers : Selection, Design and Practice,* Ann Arbor Science Publishers, Inc., Michıgan

[3] Clocksin, W.F., and Mellish, C.S., (1987), *Programming ın Prolog,* Springer-Verlag.

[4] Kelly, N.W., (1976), *Kelly's Handbook of Crossflow Coolıng Tower Performance,* Neil W. Kelly and Associates, Kansas.

[5] McKelvey, K.K., and Brooke, M., (1959), *The Industrial Cooling Tower,* Elsevier Publishing Company, New York.

[6] Mohiuddin, A.K.M., (1993), "Expert System for the Thermal Design of Wet Cooling Towers Including Database from Experiments ", *Ph.D. Thesıs,* Dept. of Mech. Engg., Indian Institute of Technology, Kanpur.

[7] Rajaram, G., and Sastry, S.S., (1994),*Project Report*, Dept. of Comp. Sc. & Engg., Indian Institute of Technology, Kanpur.

[8] Sangal, R., (1985), *Expert Systems,* CSI Communications, pp. 9–13.

[9] Sangal, R., (1991), *Programmıng Paradigms in Lisp,* McGraw Hill,Inc., pp. 119–149.

[10] Turbo Prolog (Version 2.0), *Manual*, Borland International, Inc., 1986–88.

[11] IF/Prolog (Version 3.4), *Installation Guide*, InterFace Computer GmbH, 1986–88.

APPENDIX A

Nomenclature

Vidhi An expert system shell

IF/Prolog A Prolog interpreter developed by InterFace Computer GmbH.

LISP A widely used AI language

dfs The principal predicate used in Vidhi

HP The HP 9000 computer system at IIT Kanpur

PIE The Prolog Inference Engine

PC Personal computer

askuser The built-in predicate for asking questions in Vidhi

FDs Functional dependencies

// Prolog predicate for integer division

iis Predicate implemented in the PIE for integer operations

MDCT Mechanical draft cooling tower

NDCT Natural draft cooling tower

APPENDIX B

```
[sn] vidhi
IF/Prolog Version 3.4 2 HP 9000/800 created 1/9/88
Copyright (C) 1984,88 InterFace Computer GmbH


?- [tow,ftow,ntow,usrintfc].
consult: file tow pro loaded in 2 sec.
consult. file ftow.pro loaded in 1 sec
consult: file ntow.pro loaded in 0 sec.
consult: file usrintfc pro loaded in 0 sec

yes
?- hello

 Welcome to this session  Do you need help
 regarding the use of this package ( respond with 'yes' or 'no')?  yes

 Tower is an expert system package for the selection and design of Wet Cooling
 Towers   At present, the package can help you design only the wet,
 counterflow and crossflow, natural and mechanical draft cooling towers

 Please keep in mind the following points while using the system

 1. A cooling tower is used to cool water which is used for some process.
 2  The system is designed to give a lot of options to the user.
 3  At any stage of the design, type 'help' when in doubt
 4. When typing numerical values, please do not use a comma
 5. Please use only lower case letters for any query/answer
 6  Above all, please read the questions carefully before typing your answer.
 7. Two types of questions will be asked by the system
 a. Optional questions and
 b. Mandatory questions
 An optional question may be answered by the user if he knows the value,
 otherwise type dontknow   But a mandatory question must be answered
 Otherwise the system will fail to design the tower completly.  The
 following are the Predicates of mandatory questions that might be asked
 in the system :
 installation, initial-investment, operation-and maintenance-costs,
 recirculation-and fogging, control-of-cold water-temp, approach-to-wet-bulb,
 cooling-range, inlet-water-temp, town, packing-type, total-heat-load,
 dk-type and pn-type
 8  Please type 'what' to find out the available options, for any question.

 Shall we proceed now (respond with 'yes' or 'no')?  yes

 Specify the type of cooling tower which you are interested in ? what

<This package deals with the design of two types of wet cooling towers ,
  viz   , natural and mechanical draft towers   You may indicate your choice
 by typing ' natural_draft ' or ' mechanical_draft ' . Type ' dontknow '
  otherwise   A natural draft tower is a tower in which air is introduced
 through the louvers due to the draft produced by the chimney effect .
 A mechanical draft tower is one which utilises fans to move the air
 through the tower   This gives the designer a control over the air
 supply . It may be noted that fan_assisted natural draft towers are not
 considered in this package . For selecting the type of cooling tower ,
 following factors are to be considered
 1   Is the installation a very large one , e   g   , for power plant of
 500 MWe capacity and above ?
 2 . Is the initial investment a major consideration ?
 3 . Is the total operation and maintenance costs a major consideration ?
 4   Is recirculation and fogging a major consideration ?
```

```
5   Is the strict control of cold water tempeature very important ?
6   Is a closer approach to wet bulb temperature required ?
7 . Is a longer cooling range required ?> dunno

Is the installation a large one , e g ,
                         power plant of 500 MWe capacity and above? no

Is the initial investment a major consideration ? yes

Specify your preference as to the type of flow ? crossflow

Please specify the type of packing of your preference ? type
Value has to be one of  -
        first second

Please specify the type of packing of your preference ? first

Please specify the deck type of your choice ? dk_1

Please specify the value of the ratio of water to air loading ? dunno

What is the inlet water temperature in degree Celsius ? 43

Please specify the outlet water temperature in degree Celsius? 34

Please specify the wet bulb temperature of the entering air in degree Celsius? dunno

Please specify the name of the city at which the tower is to be erected
or the nearest town amongst the names given in the table below .
You can look at the table by typing what ? lagos

Please specify the dry bulb temperature of the entering air
in degree Celsius if known ? dunno

Please specify the outlet air temperature in degree Celsius ? what

(If you have no idea as to it ' s value , please type ` dontknow '  )  dunno

Please specify the total water flow rate in kg per hour ? 450000

Specify your preference as to the type of mechanical draft? type
Value has to be one of .-
        forced_draft induced_draft

Specify your preference as to the type of mechanical draft? induced_draft

Please select and specify the air velocity through the tower
                 in meter per second? what

(You will have to specify the air velocity through the packing . For
natural draft towers it is about 1 to 2 m / s and for mechanical draft
towers it falls within the range of 1 5 to 4 m / s   If you type dontknow ,
then default value will be assumed  )  dunno

Specify the number of cycles of concentration for the tower? dunno

Specify the windage or drift losses as a percentage of total water
        circulation rate if known? dunno

Specify the evaporation losses as a percentage of total water
        circulation rate if known? dunno

Please select a type of distribution system for the tower? what

(The function of the distribution system is to spread the hot water
which requires to be cooled . There are four types in general use
```

which break up the water into fine particles , to expose as much water surface as possible to the air splash systems Two other systems are used to distribute the liquid in such a manner that it is spread in the form of a very thin film on the packing underneath , but without the the formation of droplets film flow systems .

1 Upspray_system A central header supplies the water to a net_work of pipes to which are attached the upspray nozzles . It operates at high nozzle pressure and uses large nozzle diameter which prevent clogging by algae , dirt etc It is usually located about 2 to 3 meters below the drift eliminators . It is used mainly in counterflow towers

2 . Downspray_system A central header supplies the water to a net_work of pipes to which are attached the downspray nozzles In mechanical draft towers , the system is usually located as near the top of the tower as possible and operates at a low pressure . The spray from the nozzle impinges upo a plate immediately below the nozzle opening This system does not keep itself as clean as the upspray systém , because of the lower operating pressure

3 Flume_type . This system consists of a series of troughs arranged symmetrically across the top of the tower , in the bottom of which are drilled holes The water passes through to the splash plates approximately one meter below This causes splashing of water which distributes it . This system is not as flexible as the up or downspray arrangements and requires constant maintenance

4 . Open_pan_system . This consists of a very large trough which covers effective part of the cooling system roof It is used principally in cross-flow towers A large number of small nozzles are fixed at the bottom of the pan and operate under gravity at a very low water head . These easily become plugged . It is preferable to cover the pan to prevent an accelerated rate of algae growth due to the sun ' s rays , and also to reduce the ingress of dirt .

5 Non_splash_trough_distributor A primary trough feeds the water through small vertical feed pipes into secondary troughs The water fills these to the level of narrow notches cut into the sides and spills out over the packing situated immediately below This type of distributor is used in . grid packed tower

6 Direct_feed · Incoming water is led directly on to the top of the packing plates This type is used only for certain type of film flow packings where spread of water is achieved by capillary dispersion This system suffers from clogging due to low pressure feeds , but has an easy access for cleaning the system .

Now , if you are able to select a distribution system type the corresponding number , say 4 , otherwise type dontknow > 4

Please specify the type of drift eliminator of your preference? dunno

Type of Cooling Tower Selected = mechanical_draft
Type of Flow Selected = crossflow
Wet Bulb Temperature in degree Celsius = 27.8
Dry Bulb Temperature in degree Celsius = 33.3
Inlet Water Temperature in degree Celsius = 43
Outlet Water Temperature in degree Celsius = 34
Water to Air Loading Ratio = 1 75709
Water Loading in kg/h sq m = 12000 0
Air Loading in kg/h sq m = 6829 48
Tower Characteristic Ratio = 0 357355
Type of Packing = first
Number of Decks = 3
Packed Height in meter = 1 83
Makeup Water to be Supplied in kg/h = 7366 62
Blowdown in kg/h = 1464 32
Type of drift eliminator = cellular_type_drift_eliminator
Type of Water Distribution System = open_pan_system
The Type of Mechanical Draft = induced_draft
The Total Water Flow Rate in kg/h = 450000
Tower Floor Area in sq meter = 37.5
Overall Height of the Tower in meter = 10.83
Total Static Pressure in mm of water = 3.03249

```
Total Volumetric Discharge in cubic meter/min = 17.8257
Total Fan Horse Power = 4038 15
Do you like to save these values for comparison later on ? no

Do you like to continue with another session ? no

                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                          GOODBYE
                ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Two Sessions on PC

```
-) dfs([type_of_cooling_tower(_,X)],[]).

Specify the type of cooling tower which you are interested in ? dunno

Is the installation a large one , e g.,
                              power plant of 500 MWe capacity and above? no

Is the initial investment a major consideration ? no

Is the total operation and maintenance costs a
                              a major consideration? no

Is recirculation and fogging a major consideration? no

Is the strict control of cold water temperature very important? no

Is a closer approach to wet bulb temperature required? no

Is a large cooling range required? no

X        = natural_draft

yes

-)


-) dfs([check_type_of_flow(_,X)],[])

Specify your preference as to the type of flow ? dunno

Is the recrculation of outlet air a major consideration? no

Is maintenance an important factor ? no

Is cooling per unit volume a major consideration ? no

X        = crossflow

yes

-)
```